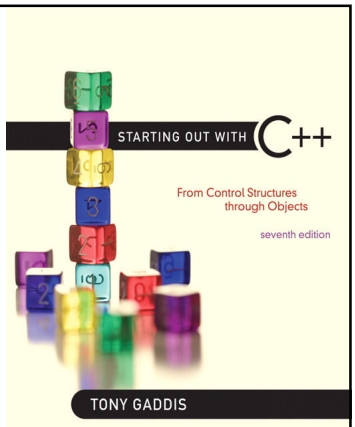
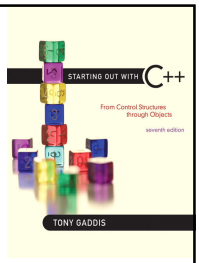
Chapter 7:

Arrays



Addison-Wesley
is an imprint of
PEARSON

7.1



Arrays Hold Multiple Values

Arrays Hold Multiple Values

- Array: variable that can store multiple values of the same type
- Values are stored in adjacent memory locations
- Declared using `[]` operator:

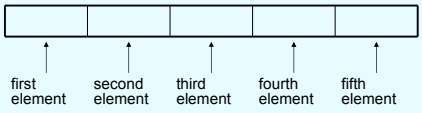
```
int tests[5];
```

Array - Memory Layout

- The definition:

```
int tests[5];
```

allocates the following memory:



Array Terminology

In the definition `int tests[5];`

- `int` is the data type of the array elements
- `tests` is the name of the array
- `5`, in `[5]`, is the size declarator. It shows the number of elements in the array.
- The size of an array is (number of elements) * (size of each element)

Array Terminology

- The size of an array is:
  - the total number of bytes allocated for it
  - (number of elements) * (number of bytes for each element)
- Examples:

  `int tests[5]` is an array of 20 bytes, assuming 4 bytes for an `int`

  `long double measures[10]` is an array of 80 bytes, assuming 8 bytes for a `long double`

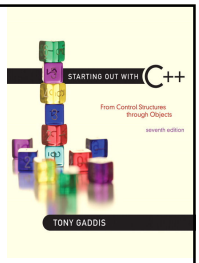
Size Declarators

- Named constants are commonly used as size declarators.

```
const int SIZE = 5;
int tests[SIZE];
```

- This eases program maintenance when the size of the array needs to be changed.

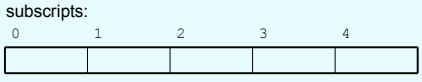
7.2



Accessing Array Elements

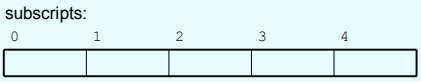
Accessing Array Elements

- Each element in an array is assigned a unique *subscript*.
- Subscripts start at 0



Accessing Array Elements

- The last element's subscript is *n*-1 where *n* is the number of elements in the array.



Accessing Array Elements

- Array elements can be used as regular variables:

```
tests[0] = 79;
cout << tests[0];
cin >> tests[1];
tests[4] = tests[0] + tests[1];
```

- Arrays must be accessed via individual elements:

```
cout << tests; // not legal
```


Program 7-1

```
1  // This program asks for the number of hours worked
2  // by six employees. It stores the values in an array.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_EMPLOYEES = 6;
9     int hours[NUM_EMPLOYEES];
10
11    // Get the hours worked by each employee.
12    cout << "Enter the hours worked by "
13         << NUM_EMPLOYEES << " employees: ";
14    cin >> hours[0];
15    cin >> hours[1];
16    cin >> hours[2];
17    cin >> hours[3];
18    cin >> hours[4];
19    cin >> hours[5];
20
```

(Program Continues)

```
21    // Display the values in the array.
22    cout << "The hours you entered are:";
23    cout << " " << hours[0];
24    cout << " " << hours[1];
25    cout << " " << hours[2];
26    cout << " " << hours[3];
27    cout << " " << hours[4];
28    cout << " " << hours[5] << endl;
29    return 0;
30 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

Here are the contents of the `hours` array, with the values entered by the user in the example output:

Accessing Array Contents

- Can access element with a constant or literal subscript:

```
cout << tests[3] << endl;
```

- Can use integer expression as subscript:

```
int i = 5;
cout << tests[i] << endl;
```

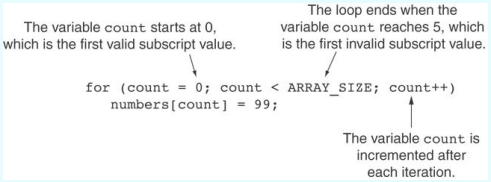

Using a Loop to Step Through an Array

- Example – The following code defines an array, `numbers`, and assigns 99 to each element:

```
const int ARRAY_SIZE = 5;
int numbers[ARRAY_SIZE];

for (int count = 0; count < ARRAY_SIZE; count++)
     numbers[count] = 99;
```

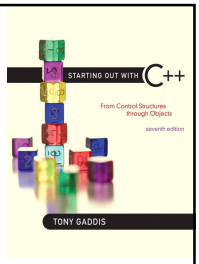
A Closer Look At the Loop



Default Initialization

- Global array → all elements initialized to 0 by default
- Local array → all elements *uninitialized* by default

7.3



No Bounds Checking in C++

No Bounds Checking in C++

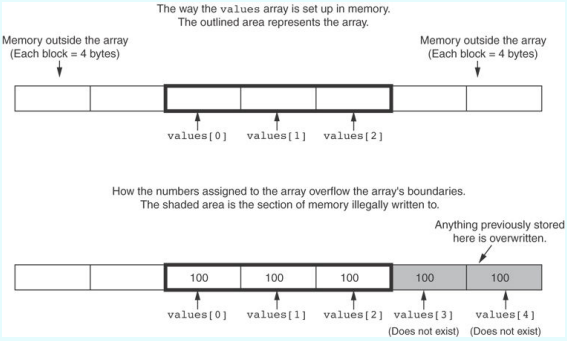
- When you use a value as an array subscript, C++ does not check it to make sure it is a *valid* subscript.
- In other words, you can use subscripts that are beyond the bounds of the array.

Code From Program 7-5

- The following code defines a three-element array, and then writes five values to it!

```
 9    const int SIZE = 3;  // Constant for the array size
10    int values[SIZE];    // An array of 3 integers
11    int count;           // Loop counter variable
12
13    // Attempt to store five numbers in the three-element array.
14    cout << "I will store 5 numbers in a 3 element array!\n";
15    for (count = 0; count < 5; count++)
16       values[count] = 100;
```


What the Code Does



No Bounds Checking in C++

- Be careful not to use invalid subscripts.
- Doing so can corrupt other memory locations, crash program, or lock up computer, and cause elusive bugs.

Off-By-One Errors

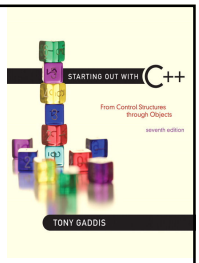
- An off-by-one error happens when you use array subscripts that are off by one.
- This can happen when you start subscripts at 1 rather than 0:

```
// This code has an off-by-one error.
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE; count++)
   numbers[count] = 0;
```


7.4

Array Initialization

7.4



Array Initialization

Array Initialization

- Arrays can be initialized with an initialization list:

```
const int SIZE = 5;
int tests[SIZE] = {79,82,91,77,84};
```

- The values are stored in the array in the order in which they appear in the list.
- The initialization list cannot exceed the array size.

Code From Program 7-6

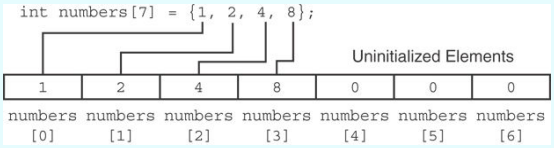
```
7     const int MONTHS = 12;
8     int days[MONTHS] = { 31, 28, 31, 30,
9                          31, 30, 31, 31,
10                         30, 31, 30, 31};
11
12    for (int count = 0; count < MONTHS; count++)
13    {
14       cout << "Month " << (count + 1) << " has ";
15       cout << days[count] << " days.\n";
16    }
```

Program Output

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

Partial Array Initialization

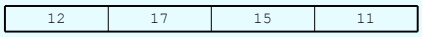
- If array is initialized with fewer initial values than the size declarator, the remaining elements will be set to `0`:



Implicit Array Sizing

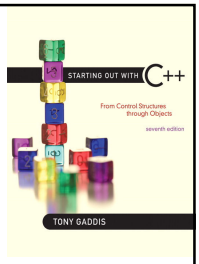
- Can determine array size by the size of the initialization list:

```
int quizzes[]={12,17,15,11};
```



- Must use either array size declarator or initialization list at array definition

7.5



Processing Array Contents

Processing Array Contents

- Array elements can be treated as ordinary variables of the same type as the array
- When using ++, -- operators, don't confuse the element with the subscript:

```
tests[i]++; // add 1 to tests[i]
tests[i++]; // increment i, no
            // effect on tests
```


Array Assignment

To copy one array to another,

- Don't try to assign one array to the other:

```
newTests = tests;  // Won't work
```

- Instead, assign element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
   newTests[i] = tests[i];
```


Printing the Contents of an Array

- You can display the contents of a *character* array by sending its name to cout:

```
char fName[] = "Henry";
cout << fName << endl;
```

But, this ONLY works with character arrays!

Printing the Contents of an Array

- For other types of arrays, you must print element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
   cout << tests[i] << endl;
```


Summing and Averaging Array Elements

- Use a simple loop to add together array elements:

```
int tnum;
double average, sum = 0;
for(tnum = 0; tnum < SIZE; tnum++)
      sum += tests[tnum];
```

- Once summed, can compute average:

```
average = sum / SIZE;
```


Finding the Highest Value in an Array

```
int count;
int highest;
highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] > highest)
      highest = numbers[count];
}
```

When this code is finished, the `highest` variable will contains the highest value in the `numbers` array.

Finding the Lowest Value in an Array

```
int count;
int lowest;
lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] < lowest)
      lowest = numbers[count];
}
```

When this code is finished, the `lowest` variable will contains the lowest value in the `numbers` array.

Partially-Filled Arrays

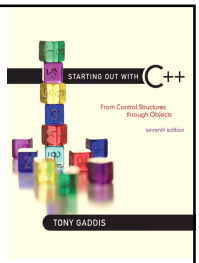
- If it is unknown how much data an array will be holding:
  - Make the array large enough to hold the largest expected number of elements.
  - Use a counter variable to keep track of the number of items stored in the array.

Comparing Arrays

- To compare two arrays, you must compare element-by-element:

```
const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0;           // Loop counter variable
// Compare the two arrays.
while (arraysEqual && count < SIZE)
{
   if (firstArray[count] != secondArray[count])
      arraysEqual = false;
   count++;
}
if (arraysEqual)
   cout << "The arrays are equal.\n";
else
   cout << "The arrays are not equal.\n";
```

7.6



Using Parallel Arrays

Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- A subscript is used to relate arrays: elements at same subscript are related
- Arrays may be of different types

Parallel Array Example

```
const int SIZE = 5;   // Array size
int id[SIZE];         // student ID
double average[SIZE]; // course average
char grade[SIZE];     // course grade
...
for(int i = 0; i < SIZE; i++)
{
   cout << "Student ID: " << id[i]
        << " average: " << average[i]
        << " grade: " << grade[i]
        << endl;
}
```

Program 7-12

```
// This program uses two parallel arrays: one for hours
// worked and one for pay rate.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int NUM_EMPLOYEES = 5;    // Number of employees
   int hours[NUM_EMPLOYEES];       // Holds hours worked
   double payRate[NUM_EMPLOYEES]; // Holds pay rates

   // Input the hours worked and the hourly pay rate.
   cout << "Enter the hours worked by " << NUM_EMPLOYEES
        << " employees and their\n"
        << "hourly pay rates.\n";
   for (int index = 0; index < NUM_EMPLOYEES; index++)
   {
      cout << "Hours worked by employee #" << (index+1) << ": ";
      cin >> hours[index];
      cout << "Hourly pay rate for employee #" << (index+1) << ": ";
      cin >> payRate[index];
   }
```

(Program Continues)

Program 7-12 *(Continued)*

```
   // Display each employee's gross pay.
   cout << "Here is the gross pay for each employee:\n";
   cout << fixed << showpoint << setprecision(2);
   for (int index = 0; index < NUM_EMPLOYEES; index++)
   {
      double grossPay = hours[index] * payRate[index];
      cout << "Employee #" << (index + 1);
      cout << ": $" << grossPay << endl;
   }
   return 0;
}
```

Program Output with Example Input Shown in Bold

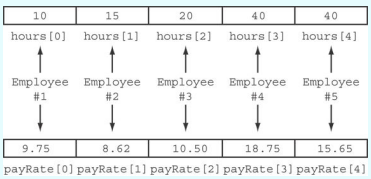
```
Enter the hours worked by 5 employees and their
hourly pay rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]
Hours worked by employee #2: 15 [Enter]
Hourly pay rate for employee #2: 8.62 [Enter]
Hours worked by employee #3: 20 [Enter]
Hourly pay rate for employee #3: 10.50 [Enter]
Hours worked by employee #4: 40 [Enter]
Hourly pay rate for employee #4: 18.75 [Enter]
Hours worked by employee #5: 40 [Enter]
Hourly pay rate for employee #5: 15.65 [Enter]
```

(program output continues)

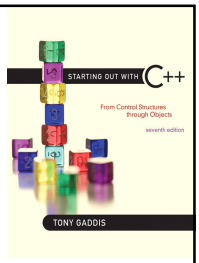
Program 7-12 *(continued)*

```
Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $129.30
Employee #3: $210.00
Employee #4: $750.00
Employee #5: $626.00
```

The `hours` and `payRate` arrays are related through their subscripts:



7.7



Arrays as Function Arguments

Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

- To define a function that takes an array parameter, use empty `[]` for array argument:

```
void showScores(int []);
            // function prototype
void showScores(int tests[])
            // function header
```


Arrays as Function Arguments

- When passing an array to a function, it is common to pass array size so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

```
void showScores(int [], int);
            // function prototype
void showScores(int tests[], int size)
            // function header
```

Program 7-14

```
1  // This program demonstrates an array being passed to a function.
2  #include <iostream>
3  using namespace std;
4
5  void showValues(int [], int); // Function prototype
6
7  int main()
8  {
9     const int ARRAY_SIZE = 8;
10    int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12    showValues(numbers, ARRAY_SIZE);
13    return 0;
14 }
15
```

(Program Continues)

Program 7-14 *(Continued)*

```
16 //************************************************
17 // Definition of function showValue.             *
18 // This function accepts an array of integers and *
19 // the array's size as its arguments. The contents *
20 // of the array are displayed.                    *
21 //************************************************
22
23 void showValues(int nums[], int size)
24 {
25    for (int index = 0; index < size; index++)
26       cout << nums[index] << " ";
27    cout << endl;
28 }
```

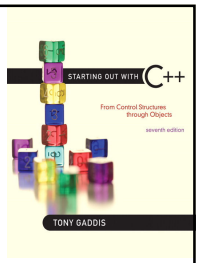
Program Output

```
5 10 15 20 25 30 35 40
```


Modifying Arrays in Functions

- Array names in functions are like reference variables – changes made to array in a function are reflected in actual array in calling function
- Need to exercise caution that array is not inadvertently changed by a function

7.8



Two-Dimensional Arrays

Two-Dimensional Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition:

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

- First declarator is number of rows; second is number of columns

Two-Dimensional Array Representation

```
const int ROWS = 4, COLS = 3; int
exams[ROWS][COLS];
```

columns

rows

exams[0][0]	exams[0][1]	exams[0][2]
exams[1][0]	exams[1][1]	exams[1][2]
exams[2][0]	exams[2][1]	exams[2][2]
exams[3][0]	exams[3][1]	exams[3][2]

- Use two subscripts to access element:

```
exams[2][2] = 86;
```

Program 7-18

```
// This program demonstrates a two-dimensional array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int NUM_DIVS = 3;           // Number of divisions
   const int NUM_QTRS = 4;           // Number of quarters
   double sales[NUM_DIVS][NUM_QTRS]; // Array with 3 rows and 4 columns.
   double totalSales = 0;            // To hold the total sales.
   int div, qtr;                     // Loop counters.

   cout << "This program will calculate the total sales of\n";
   cout << "all the company's divisions.\n";
   cout << "Enter the following sales information:\n\n";

```

(program continues)

Program 7-18 *(continued)*

```
   // Nested loops to fill the array with quarterly
   // sales figures for each division.
   for (div = 0; div < NUM_DIVS; div++)
   {
      for (qtr = 0; qtr < NUM_QTRS; qtr++)
      {
         cout << "Division " << (div + 1);
         cout << ", Quarter " << (qtr + 1) << ": $";
         cin >> sales[div][qtr];
      }
      cout << endl; // Print blank line.
   }

   // Nested loops used to add all the elements.
   for (div = 0; div < NUM_DIVS; div++)
   {
      for (qtr = 0; qtr < NUM_QTRS; qtr++)
         totalSales += sales[div][qtr];
   }

   cout << fixed << showpoint << setprecision(2);
   cout << "The total sales for the company are: $";
   cout << totalSales << endl;
   return 0;
}
```


Program Output with Example Input Shown in Bold

```
This program will calculate the total sales of
all the company's divisions.
Enter the following sales data:

Division 1, Quarter 1: $31569.45 [Enter]
Division 1, Quarter 2: $29654.23 [Enter]
Division 1, Quarter 3: $32982.54 [Enter]
Division 1, Quarter 4: $39651.21 [Enter]

Division 2, Quarter 1: $56321.02 [Enter]
Division 2, Quarter 2: $54128.63 [Enter]
Division 2, Quarter 3: $41235.85 [Enter]
Division 2, Quarter 4: $54652.33 [Enter]

Division 3, Quarter 1: $29654.35 [Enter]
Division 3, Quarter 2: $28963.32 [Enter]
Division 3, Quarter 3: $25353.55 [Enter]
Division 3, Quarter 4: $32615.88 [Enter]

The total sales for the company are: $456782.34
```

2D Array Initialization

- Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = { {84, 78},
                          {92, 97} };
```

84	78
92	97

- Can omit inner `{ }`, some initial values in a row – array elements without initial values will be set to `0` or `NULL`

Two-Dimensional Array as Parameter, Argument

- Use array name as argument in function call:

```
getExams(exams, 2);
```

- Use empty `[]` for row, size declarator for column in prototype, header:

```
const int COLS = 2;
// Prototype
void getExams(int [][COLS], int);

// Header
void getExams(int exams[][COLS], int rows)
```


Example – The `showArray` Function from Program 7-19

```
30 //*************************************************************
31 // Function Definition for showArray                          *
32 // The first argument is a two-dimensional int array with COLS *
33 // columns. The second argument, rows, specifies the number of *
34 // rows in the array. The function displays the array's contents. *
35 //*************************************************************
36
37 void showArray(int array[][COLS], int rows)
38 {
39    for (int x = 0; x < rows; x++)
40    {
41       for (int y = 0; y < COLS; y++)
42       {
43          cout << setw(4) << array[x][y] << " ";
44       }
45       cout << endl;
46    }
47 }
```

How `showArray` is Called

```
15     int table1[TBL1_ROWS][COLS] = {{1, 2, 3, 4},
16                                    {5, 6, 7, 8},
17                                    {9, 10, 11, 12}};
18     int table2[TBL2_ROWS][COLS] = {{10, 20, 30, 40},
19                                    {50, 60, 70, 80},
20                                    {90, 100, 110, 120},
21                                    {130, 140, 150, 160}};
22
23     cout << "The contents of table1 are:\n";
24     showArray(table1, TBL1_ROWS);
25     cout << "The contents of table2 are:\n";
26     showArray(table2, TBL2_ROWS);
```


Summing All the Elements in a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_ROWS = 5; // Number of rows
const int NUM_COLS = 5; // Number of columns
int total = 0;          // Accumulator
int numbers[NUM_ROWS][NUM_COLS] =
   {{2, 7, 9, 6, 4},
    {6, 1, 8, 9, 4},
    {4, 3, 7, 2, 9},
    {9, 9, 0, 3, 1},
    {6, 2, 7, 4, 1}};
```


Summing All the Elements in a Two-Dimensional Array

```
// Sum the array elements.
for (int row = 0; row < NUM_ROWS; row++)
{
   for (int col = 0; col < NUM_COLS; col++)
      total += numbers[row][col];
}

// Display the sum.
cout << "The total is " << total << endl;
```

Summing the Rows of a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
     {{88, 97, 79, 86, 94},
      {86, 91, 78, 79, 84},
      {82, 73, 77, 82, 89}};
```


Summing the Rows of a Two-Dimensional Array

```
// Get each student's average score.
for (int row = 0; row < NUM_STUDENTS; row++)
{
   // Set the accumulator.
   total = 0;
   // Sum a row.
   for (int col = 0; col < NUM_SCORES; col++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_SCORES;
   // Display the average.
   cout << "Score average for student "
        << (row + 1) << " is " << average <<endl;
}
```


Summing the Columns of a Two-Dimensional Array

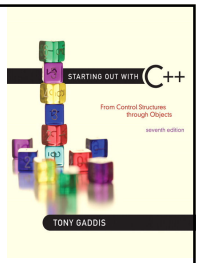
- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
     {{88, 97, 79, 86, 94},
      {86, 91, 78, 79, 84},
      {82, 73, 77, 82, 89}};
```

Summing the Columns of a Two-Dimensional Array

```
// Get the class average for each score.
for (int col = 0; col < NUM_SCORES; col++)
{
   // Reset the accumulator.
   total = 0;
   // Sum a column
   for (int row = 0; row < NUM_STUDENTS; row++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_STUDENTS;
   // Display the class average.
   cout << "Class average for test " << (col + 1)
        << " is " << average << endl;
}
```


7.9



Arrays with Three or More Dimensions

Arrays with Three or More Dimensions

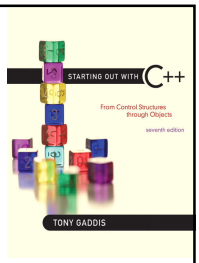
- Can define arrays with any number of dimensions:

```
short rectSolid[2][3][5];
double timeGrid[3][4][3][4];
```

- When used as parameter, specify all but 1st dimension in prototype, heading:

```
void getRectSolid(short [][3][5]);
```

7.11



Introduction to the STL `vector`

Introduction to the STL `vector`

- A data type defined in the Standard Template Library (covered more in Chapter 16)
- Can hold values of any type:

```
vector<int> scores;
```

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elements

Declaring Vectors

- You must `#include<vector>`
- Declare a vector to hold `int` element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

Adding Elements to a Vector

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```


Removing Vector Elements

- Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

- To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```


Other Useful Member Functions

Member Function	Description	Example
`at(elt)`	Returns the value of the element at position `elt` in the vector	`cout << vec1.at(i);`
`capacity()`	Returns the maximum number of elements a vector can store without allocating more memory	`maxelts = vec1.capacity();`
`reverse()`	Reverse the order of the elements in a vector	`vec1.reverse();`
`resize (elts,val)`	Add elements to a vector, optionally initializes them	`vec1.resize(5,0);`
`swap(vec2)`	Exchange the contents of two vectors	`vec1.swap(vec2);`